# Mini Project – Neural Network – Report


## מגישים:

- אריאל הרטל – 324075035
- נועה גולן – 206846966
- מתניה קנינו - 208212381


## Part I: the classifier and optimizer

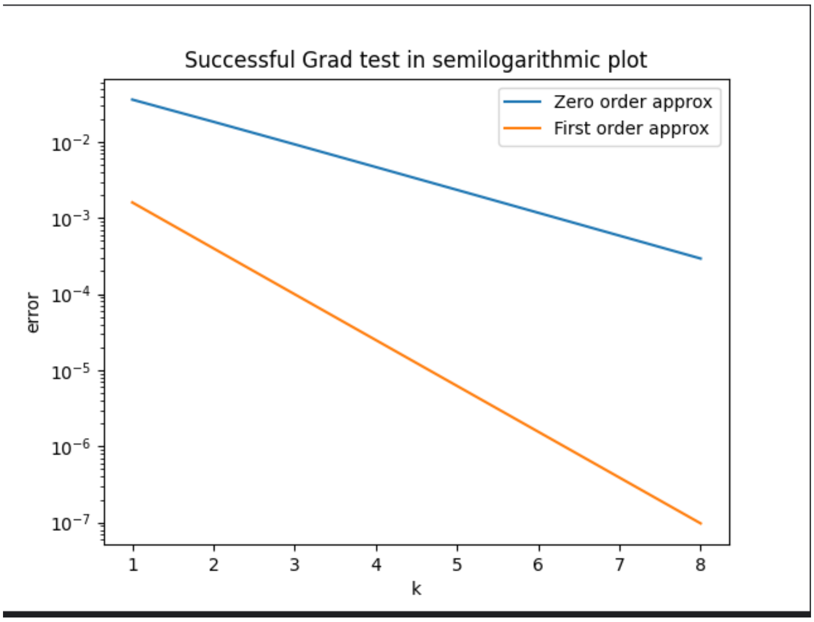
### Methods:

- **net_input -** This function calculates the dot product of the weights W and the input data X.
- **softmax –** calculating the soft-max regression of given output of a network.
- **compute_loss** - This function computes the loss of the softmax regression model.
- **compute_gradient** - This function computes the gradient of the loss function with respect to the weights.
- **gradient_test** - This function checks if the gradient of the loss function is computed correctly.
- **sgd_with_momentum_least_squares** - This function uses Stochastic Gradient Descent (SGD) with momentum to find the best weights for a least squares problem.
- **least_squares_loss** - This function calculates the mean squared error between the predicted and actual values.
- **least_squares_gradient** - This function calculates the gradient of the least squares loss.
- **sgd_with_momentum** - This function uses Stochastic Gradient Descent (SGD) with momentum to find the best weights for a given problem.
- **train_sgd_with_momentum** - This function trains a model using Stochastic Gradient Descent (SGD) with momentum.
- **compute_accuracy** - This function calculates the accuracy of the model's predictions.

## 2.1.1 – Gradient test for soft-max regression

### Input 1:

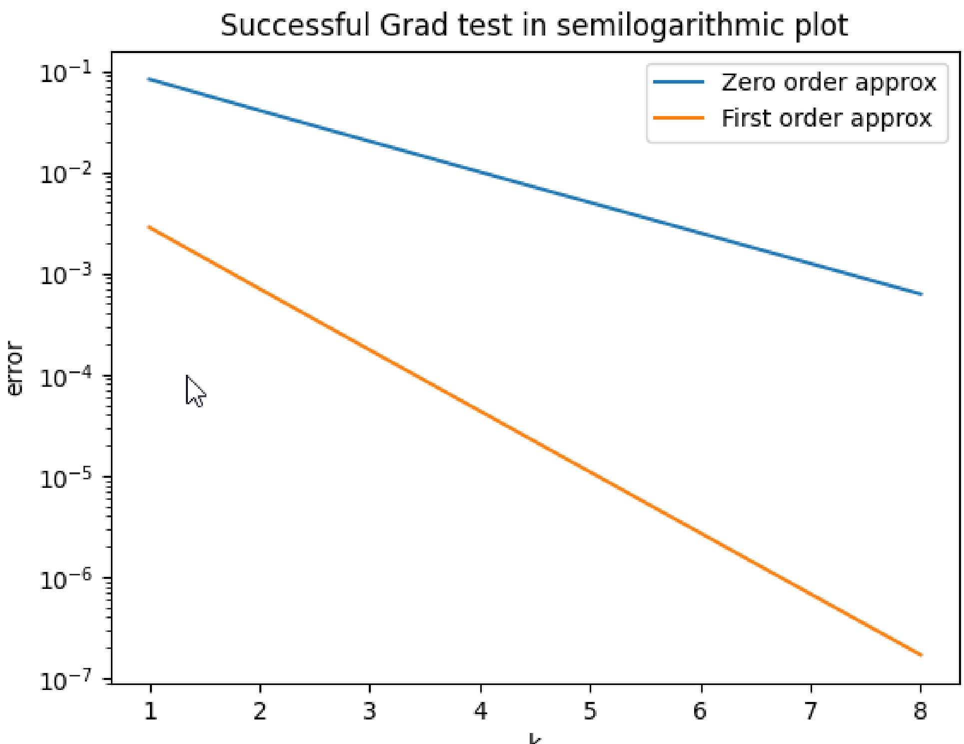
**mat_data1 = loadmat('GMMData.mat')**
X_train = mat_data1['Yt']
C_train = mat_data1['Ct']




| k | error order 1 | error order 2 |
|---|---|---|
| 1 | 0.035975022390158706 | 0.0016041066980601038 |
| 2 | 0.01838989346552511 | 0.000399671078584074 |
| 3 | 0.00929503934860998 | 9.974292344461233e-05 |
| 4 | 0.004672477568764322 | 2.4913567262974112e-05 |
| 5 | 0.002342469964957683 | 6.2256030561869125e-06 |
| 6 | 0.0011727917329524828 | 1.5560510542300676e-06 |
| 7 | 0.0005867849230214439 | 3.889689819125408e-07 |
| 8 | 0.00029348970923281925 | 9.723676885897703e-08 |

# Input 2:

**mat_data2 = loadmat('PeaksData.mat')**
X_train = mat_data1['Yt']
C_train = mat_data1['Ct']



| k | error order 1 | error order 2 |
|---|---|---|
| 1 | 0.08264306279846467 | 0.0028428907038389184 |
| 2 | 0.04060292058360071 | 0.000702834536288055 |
| 3 | 0.02012470621267548 | 0.000174663189019153 |
| 4 | 0.010018553316175627 | 4.3531804347463066e-05 |
| 5 | 0.004998376763857948 | 1.0866007943644007e-05 |
| 6 | 0.0024964697503344624 | 2.71437237753247e-06 |
| 7 | 0.0012475560151523624 | 6.783261738974034e-07 |
| 8 | 0.0006236083926229874 | 1.695481337549154e-07 |

## 2.1.2 – SGD small least squares example

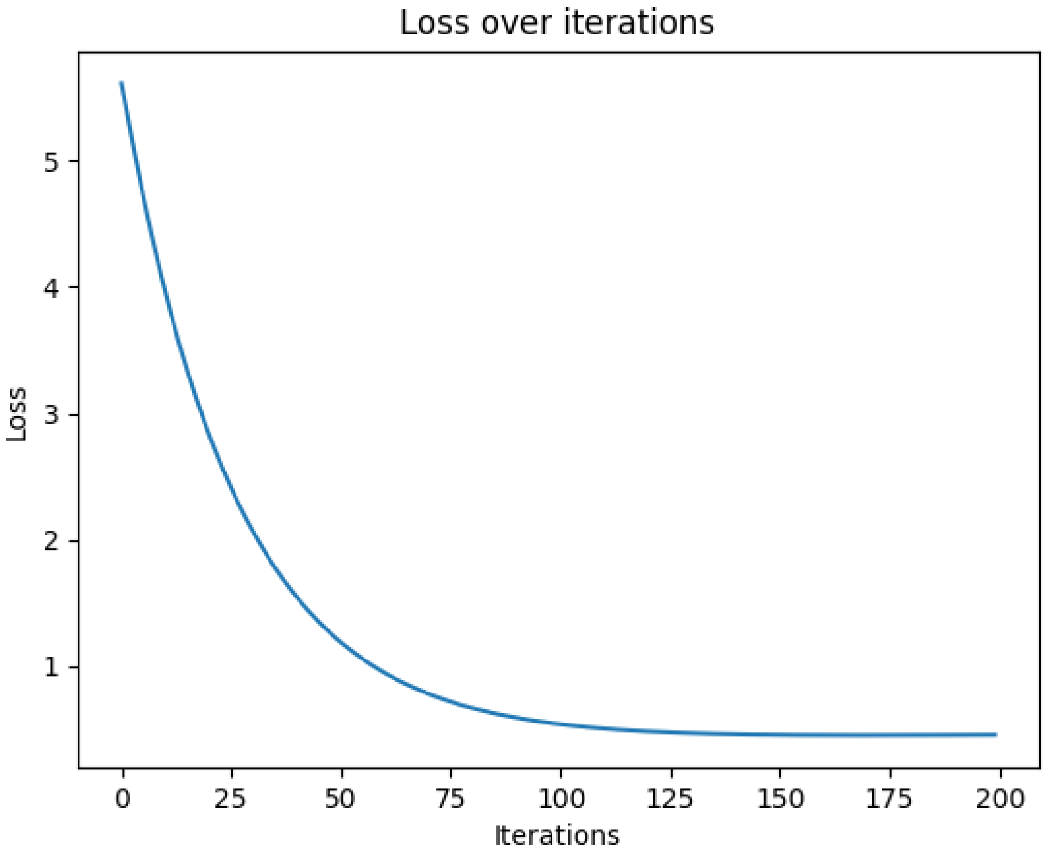
## Input:

learning_rate=0.02
momentum=0.1
num_iterations=200
X = np.random.rand(10, 1)
y = 2 * X + 1 + 0.1 * np.random.randn(10, 1)

```
def sgd_with_momentum_least_squares(obj_func, grad_func, X, y, learning_rate=0.02, momentum=0.1, num_iterations=200):
    weights = np.random.randn(X.shape[0], y.shape[1])
    velocity = np.zeros_like(weights)
    losses = []
    for i in range(num_iterations):
        # Compute objective function and gradient
        loss = obj_func(X, weights, y)
        gradient = grad_func(weights)
        # Update weights using SGD with momentum
        velocity = momentum * velocity + learning_rate * gradient
        weights -= velocity

        losses.append(loss)

    return weights, losses
```




## 2.1.3 – SGD for softmax – regression

**Try 1:**

**Input:**

learning_rate = 0.1
batch_size=5

The accuracy for both training data and validation data are getting bigger at a linear rate.

**Try 2:**

**Input:**

learning_rate = 0.02
batch_size=10

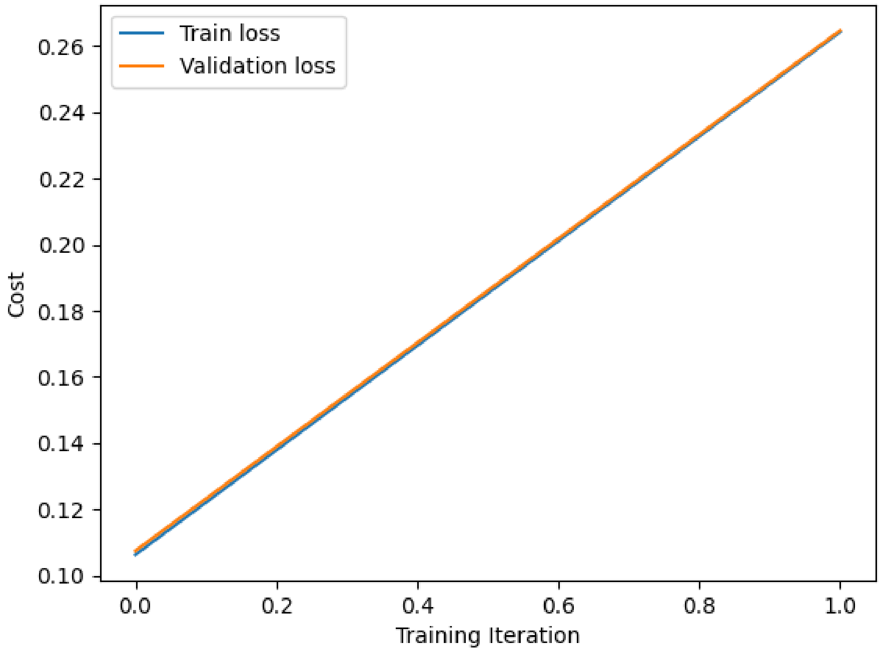
The accuracy for both training data and validation data are getting bigger at a linear rate, and are almost identical (almost merging).

## Try 3:

## Input:

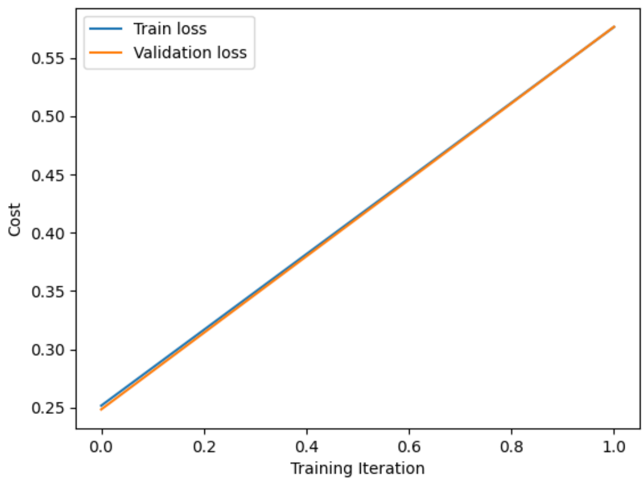
learning_rate = 0.001
batch_size=15

Same as before (linear rate), but are even more close to be merged together.




---

If we do the same tries but with “PeaksData.mat” (same input for all 3 tries), we get similar results.
**This is what we get for try 3 for “PeaksData.mat”:**



# Part II: the neural network

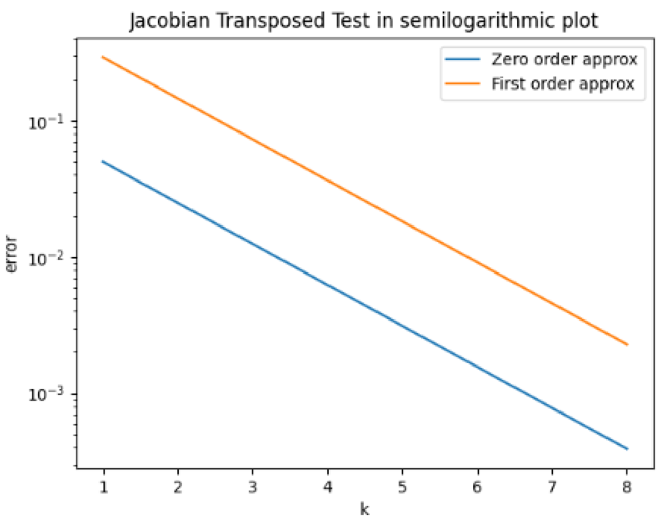
## Methods:

- **init_net** - This function initializes the weights and biases for each layer in your neural network.
- **forward_lin** - This function computes the linear part of a layer's forward propagation step.
- **forward_act** - This function computes the forward propagation for the layer by the activation function.
- **forward_net_model** - This function implements forward propagation for the entire model.
- **backward_lin** - This function computes the gradient of the cost with respect to the activation, weights, and biases.
- **backward_act** - This function computes the backward propagation for the layer by the activation function.
- **softmax_backward** - This function computes the gradient of the cost with respect to the network ouput for softmax activation.
- **backward_net_model** - This function implements the backward propagation for the entire model.
- **param_update** - This function updates the parameters using gradient descent.
- **jacobian_transposed_test** - This function performs a Jacobian transposed test.
- **net_model** - This function trains a deep learning model with multiple layers.
- **calculate_accuracy** - the function calculates the accuracy of the model's predictions.

# 2.2.1 – Jacobian test for layers

# Input:

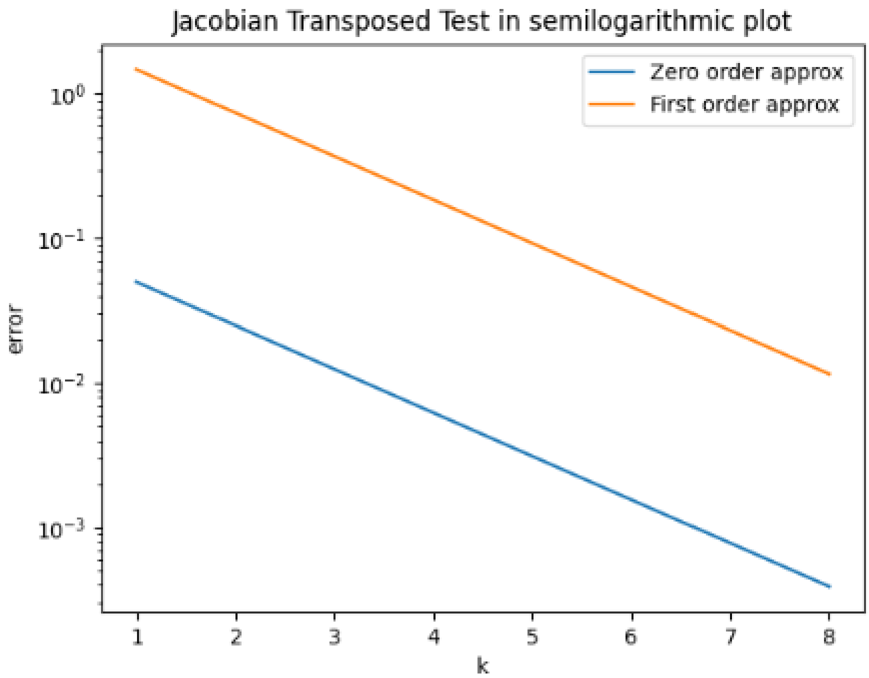
X = np.random.randn(10, 100)
Y = np.random.randn(10, 100)
layer_dims = [10, 5, 3, 10]



| k | error order 1 | error order 2 |
|---|---|---|
| 1 | 0.05000000000000002 | 0.2925077088420609 |
| 2 | 0.025000000000000005 | 0.14625385442103042 |
| 3 | 0.012500000000000002 | 0.07312692721051521 |
| 4 | 0.006250000000000002 | 0.036563463605257605 |
| 5 | 0.0031250000000000028 | 0.018281731802628803 |
| 6 | 0.001562500000000001 | 0.0091408659013144 |
| 7 | 0.0007812499999999979 | 0.004570432950657201 |
| 8 | 0.0003906249999999992 | 0.0022852164753286003 |

## 2.2.2 – Jacobian test for layers – ResNet

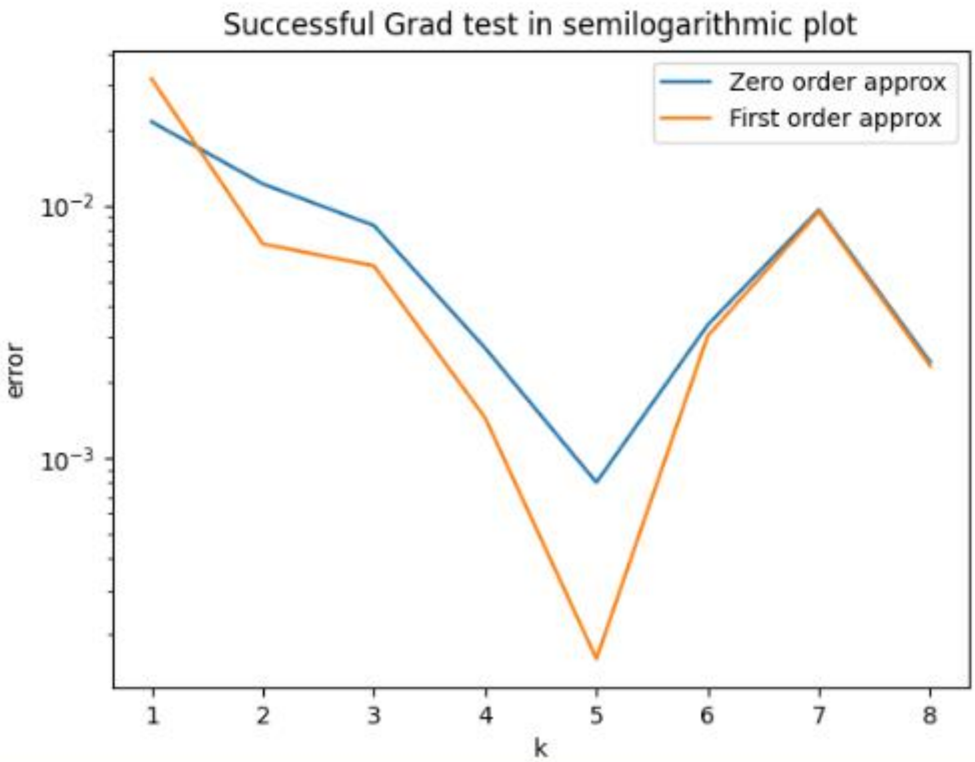
(Input same as before)



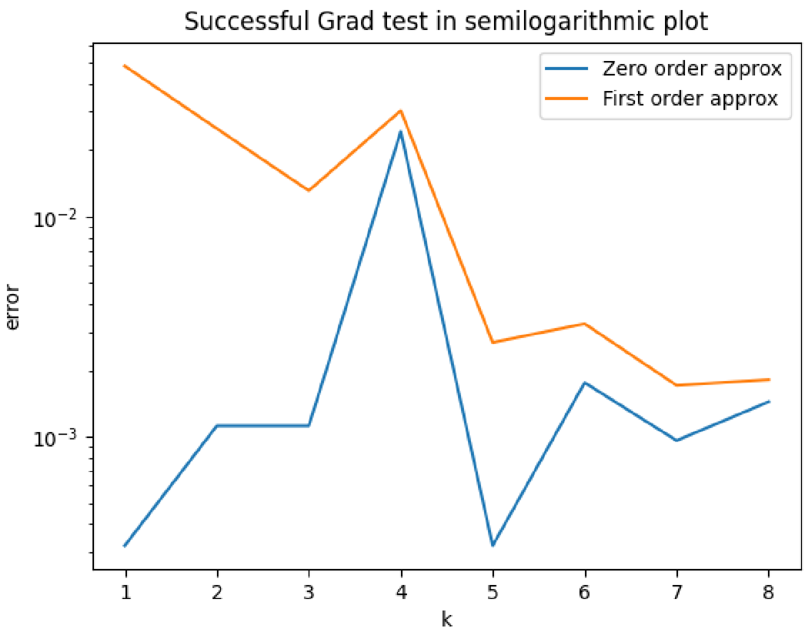

| k | error order 1 | error order 2 |
|---|---|---|
| 1 | 0.05000000000000002 | 1.483056650176104 |
| 2 | 0.024999999999999908 | 0.741528325088052 |
| 3 | 0.012499999999999805 | 0.37076416254402605 |
| 4 | 0.006250000000000092 | 0.18538208127201297 |
| 5 | 0.0031250000000001953 | 0.0926910406360065 |
| 6 | 0.0015624999999999077 | 0.046345520318003264 |
| 7 | 0.0007812499999998043 | 0.0231727601590016 |
| 8 | 0.00039062500000009294 | 0.011586380079500825 |

## 2.2.3 – Grad test for whole network

**For "GMMData.mat":**




**For "PeaksData.mat":**



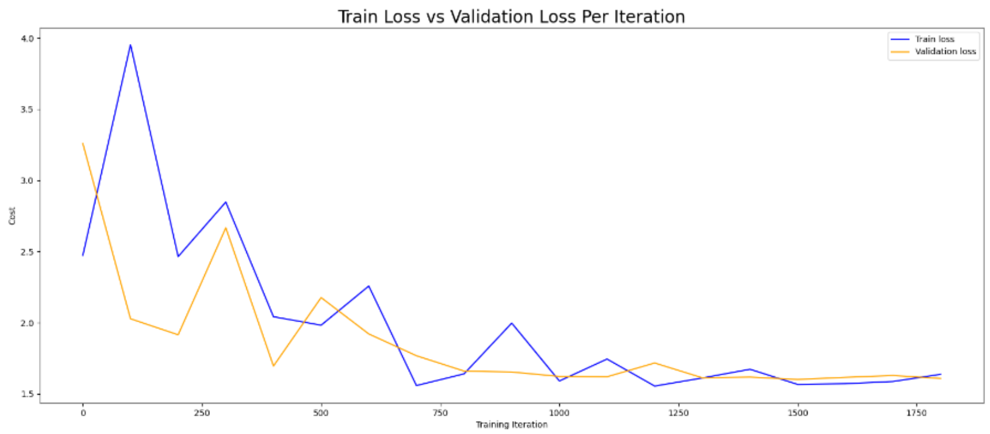
# 2.2.4 – SGD for softmax – whole network

## Input:

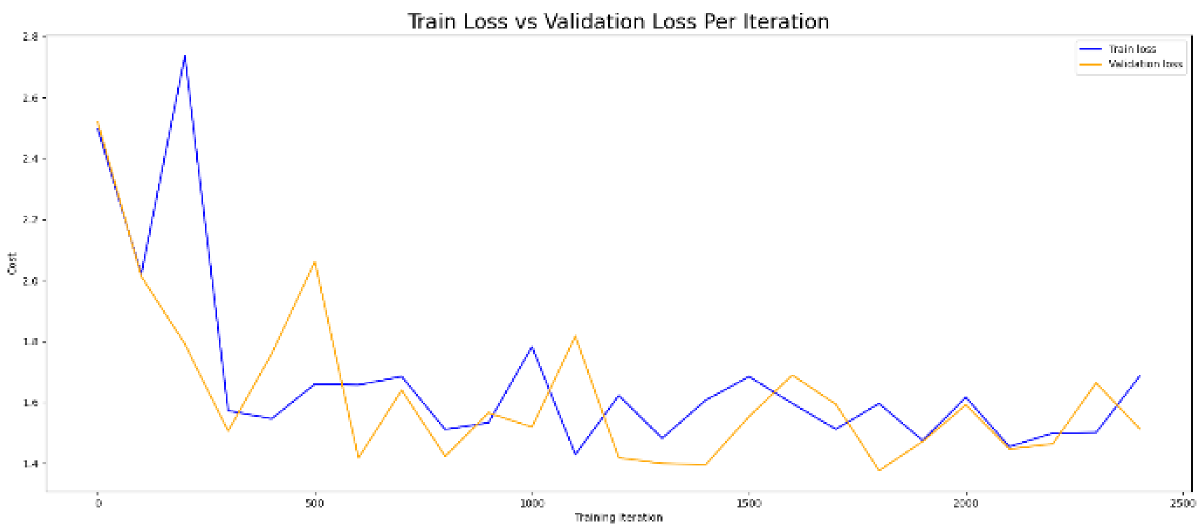
## Try 1:

layer_dims = [X_train.shape[0], 2, 3, 4, 5]
learning_rate = 0.009
num_iterations = 100
batch_size = 32

**For "GMMData.mat":**



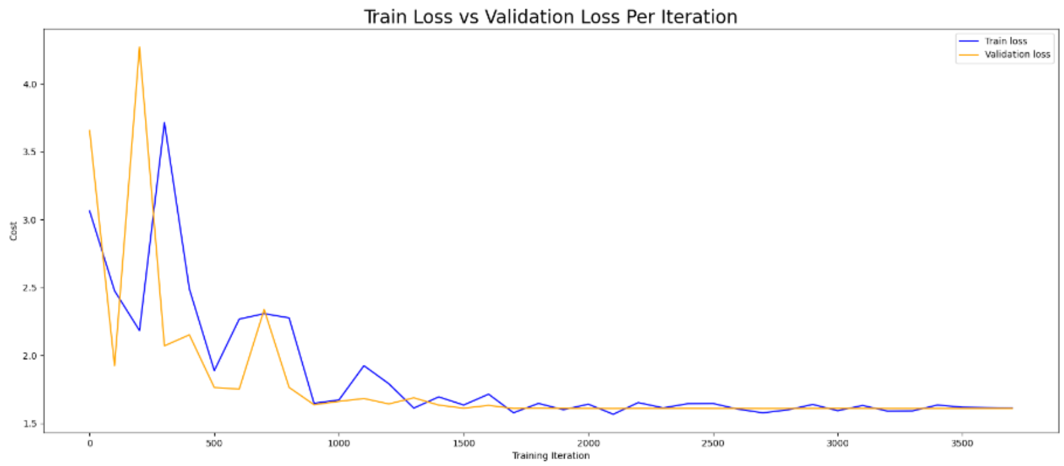

**For "PeaksData.mat":**



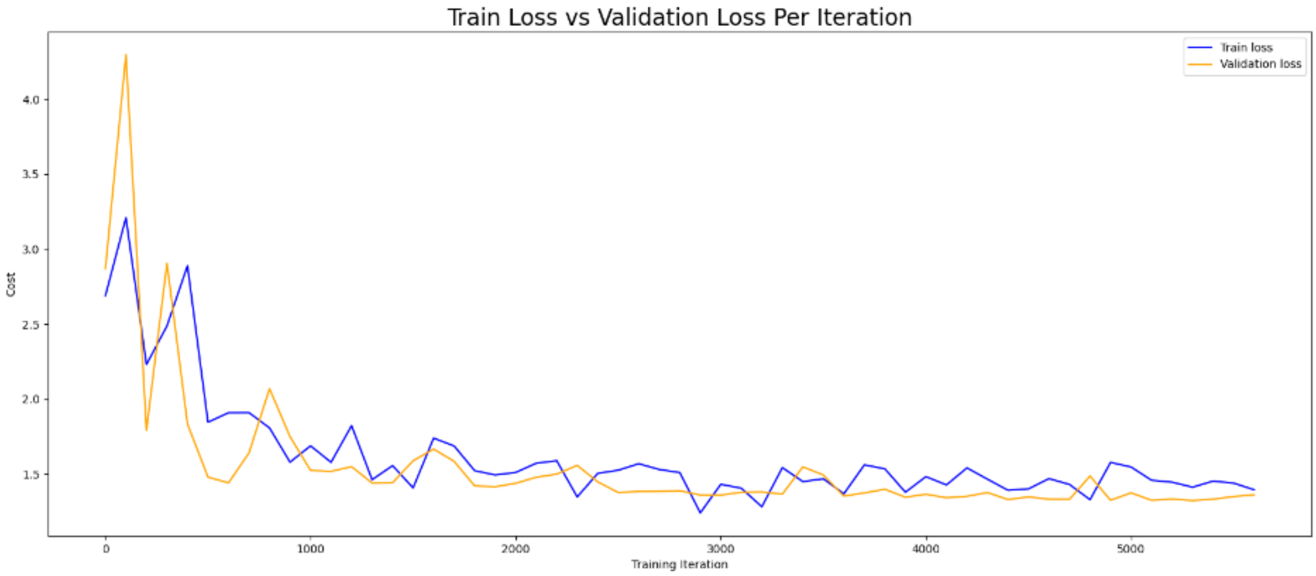
# Try 2:

layer_dims = [X_train.shape[0], 2, 2, 2,  3, 4, 5]
learning_rate = 0.009
num_iterations = 100
batch_size = 32

**For “GMMData.mat”:**



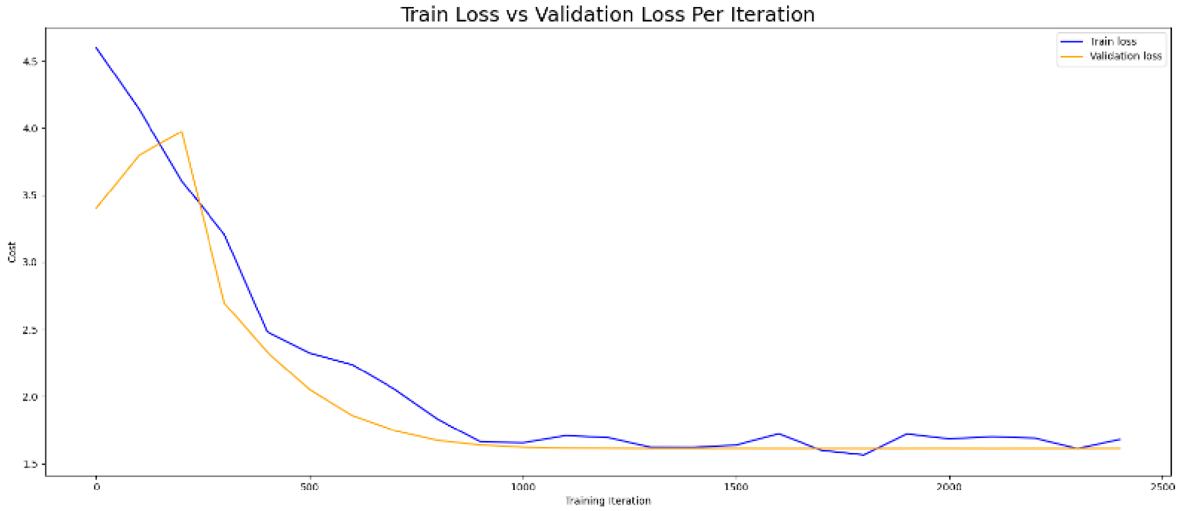

**For “PeaksData.mat”:**



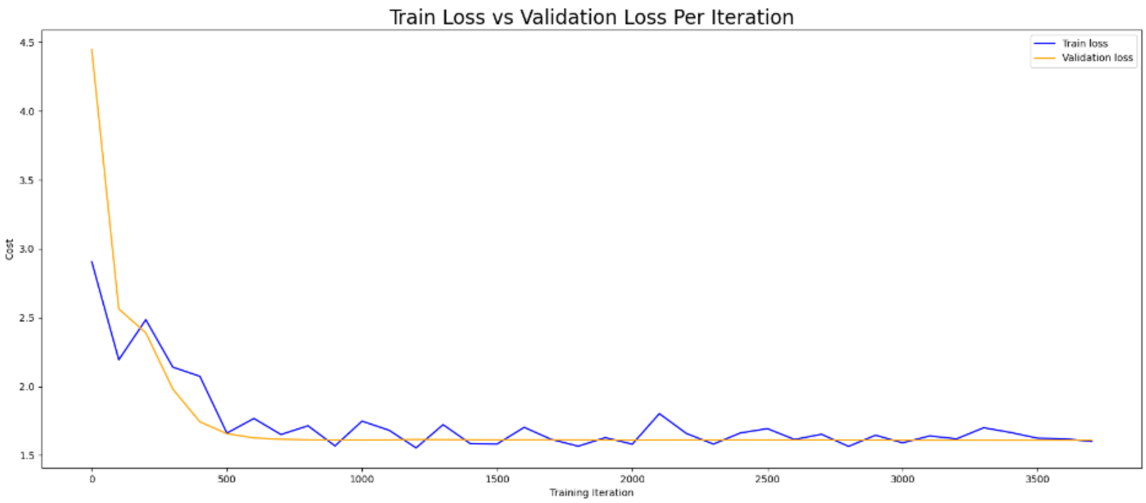
# Try 3:

layer_dims = [X_train.shape[0], 3, 2, 2, 4, 2,  2, 5]
learning_rate = 0.009
num_iterations = 100
batch_size = 32

### For “GMMData.mat”:




### For “PeaksData.mat”:



## Conclusions:

When we add more layers to our network, it gets better at learning from our training data. This means it can also do a better job on the validation data, which it hasn’t seen before. That’s why the difference between the training and validation results gets smaller much quicker as the net is longer.
But if the network too big, it can do good on the training data (by learning it) but do bad on the validation data.

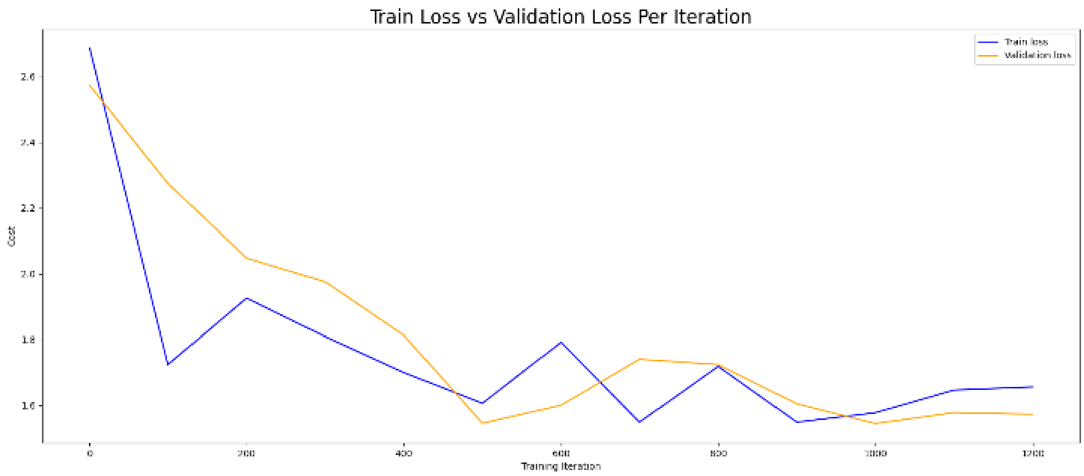
# 2.2.5 – Minimize expense of NNs

# Input:

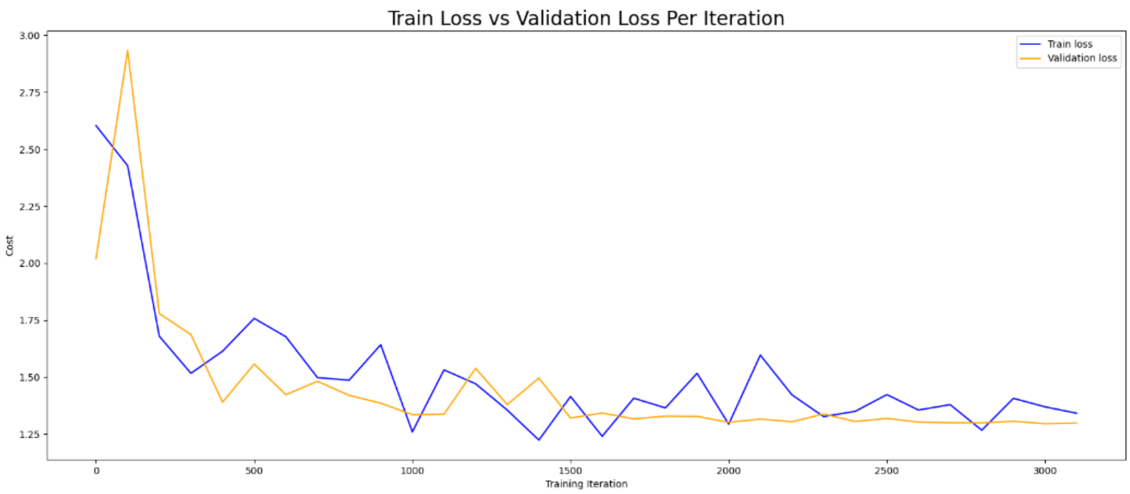
# Try 1:

layer_dims = [X_train.shape[0], 20, 7, 15, 5]
learning_rate = 0.009
num_iterations = 100
batch_size = 32

### For “GMMData.mat”:



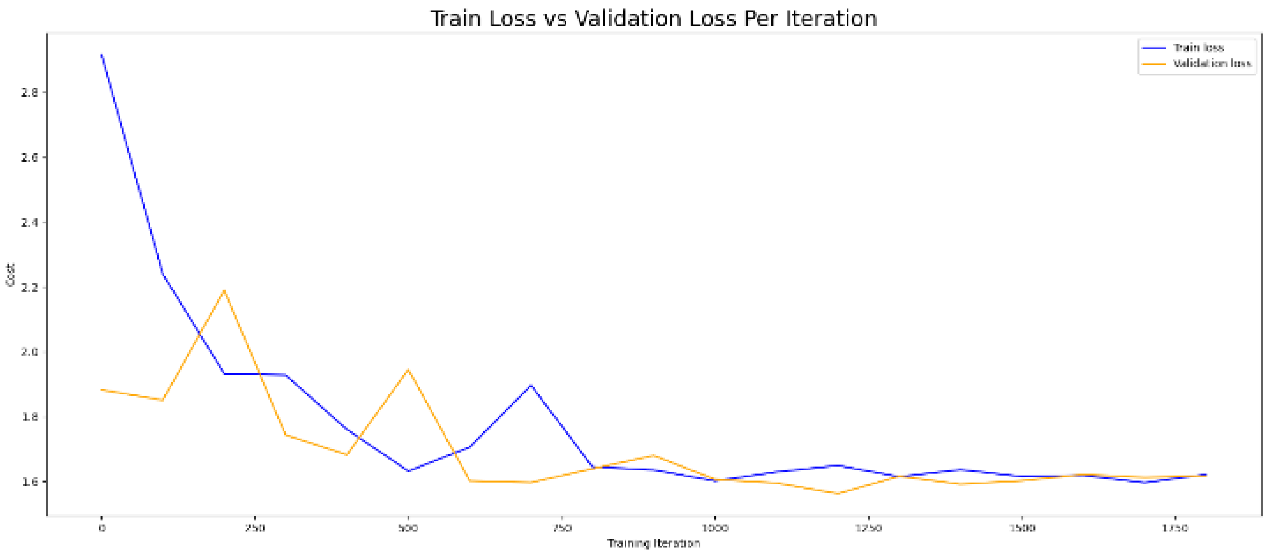

### For “PeaksData.mat”:



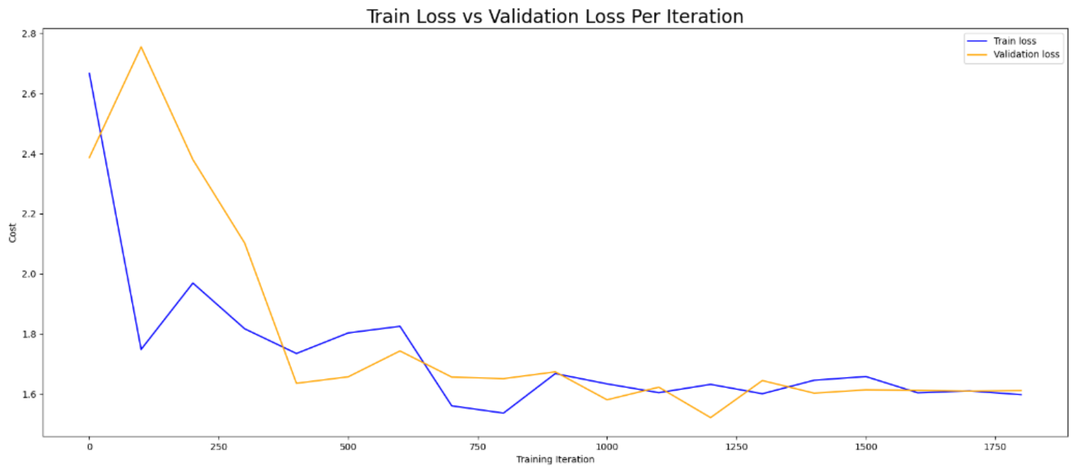
# Try 2:

layer_dims = [X_train.shape[0], 20, 4, 7, 2, 3, 15, 5]
learning_rate = 0.009
num_iterations = 100
batch_size = 32

**For “GMMData.mat”:**



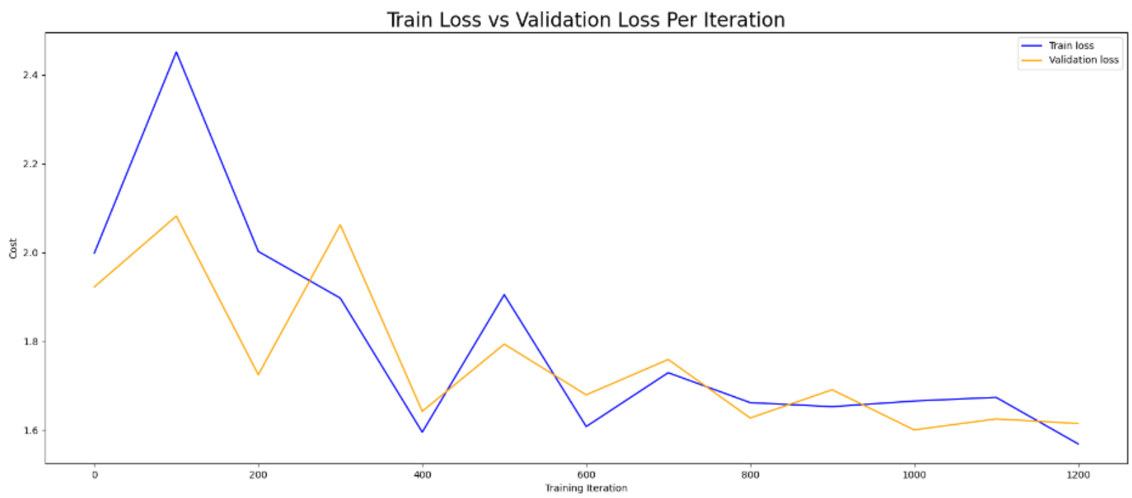

**For “PeaksData.mat”:**



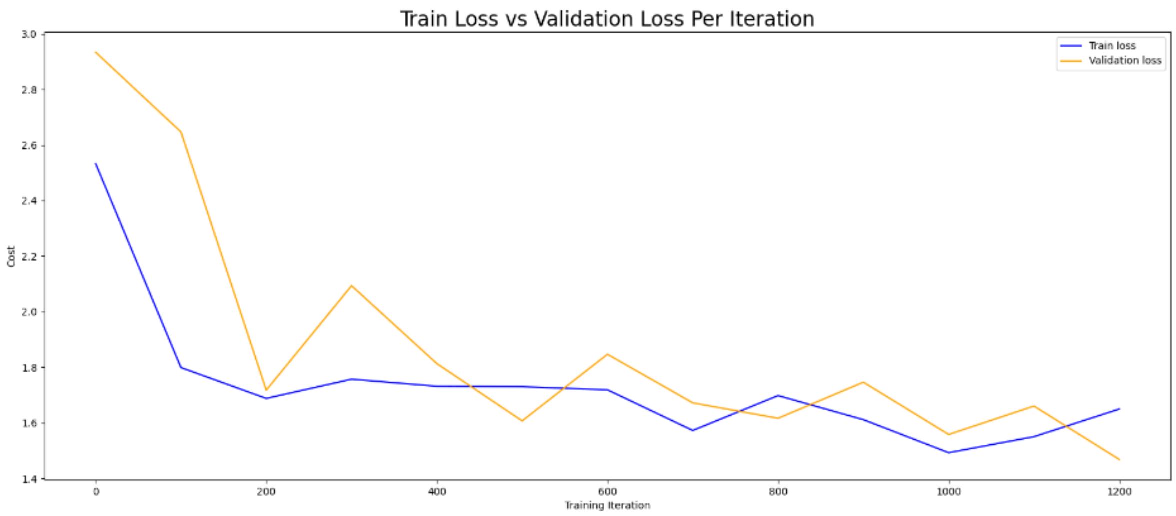

# Try 3:

layer_dims = [X_train.shape[0], 10, 8, 2, 4, 7,  15, 5]
learning_rate = 0.009
num_iterations = 100
batch_size = 32

**For “GMMData.mat”:**




**For “PeaksData.mat”:**




# Explanation:

# Try 1 got - Test accuracy: 32.5760%
# Try 2 got - Test accuracy: 11.6000%
# Try 3 got - Test accuracy: 20.0480%

In our tests, we found that using fewer layers with more neurons in each layer worked best for our problem. This might be because our data doesn’t need a complex structure to be understood well. Instead, having more neurons in each layer helped the network understand the data better.
Also, having more neurons but fewer layers helped us keep the total number of parameters be less than 100C (~ 500) This is important because we want our network to be efficient and not use more resources than necessary.

1. Try 1 - layer_dims = [X_train.shape[0], 20, 7, 15, 5]:

    Input Layer to First Hidden Layer: 5 * 20 + 20 = 120 parameters

    First Hidden Layer to Second Hidden Layer: 20 * 7 + 7 = 147 parameters

    Second Hidden Layer to Third Hidden Layer: 7 * 15 + 15 = 120 parameters

    Third Hidden Layer to Output Layer: 15 * 5 + 5 = 80 parameters

   Total 467 parameters

2. Try 2 - layer_dims = [X_train.shape[0], 20, 4, 7, 2, 3, 15, 5]:

   Input Layer to First Hidden Layer: 5 * 20 + 20 = 120 parameters

   First Hidden Layer to Second Hidden Layer: 20 * 4 + 4 = 84 parameters

   Second Hidden Layer to Third Hidden Layer: 4 * 7 + 7 = 35 parameters

   Third Hidden Layer to Fourth Hidden Layer: 7 * 2 + 2 = 16 parameters

   Fourth Hidden Layer to Fifth Hidden Layer: 2 * 3 + 3 = 9 parameters

   Fifth Hidden Layer to Sixth Hidden Layer: 3 * 15 + 15 = 60 parameters

   Sixth Hidden Layer to Output Layer: 15 * 5 + 5 = 80 parameters

   Total 404 parameters.

3. **Try 3** - layer_dims = [X_train.shape[0], 10, 8, 2, 4, 7, 15, 5]:

   Input Layer to First Hidden Layer: 5 * 10 + 10 = 60 parameters

   First Hidden Layer to Second Hidden Layer: 10 * 8 + 8 = 88 parameters

   Second Hidden Layer to Third Hidden Layer: 8 * 2 + 2 = 18 parameters

   Third Hidden Layer to Fourth Hidden Layer: 2 * 4 + 4 = 12 parameters

   Fourth Hidden Layer to Fifth Hidden Layer: 4 * 7 + 7 = 35 parameters

   Fifth Hidden Layer to Sixth Hidden Layer: 7 * 15 + 15 = 120 parameters

   Sixth Hidden Layer to Output Layer: 15 * 5 + 5 = 80 parameters

   Total 413 parameters